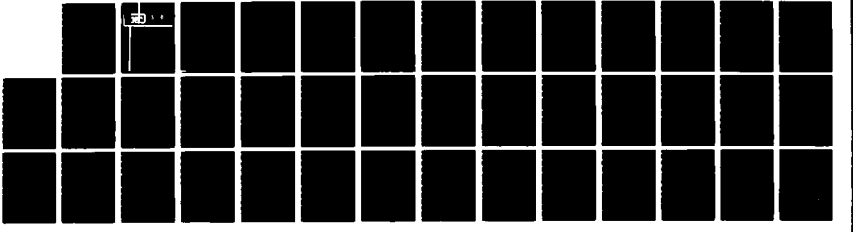
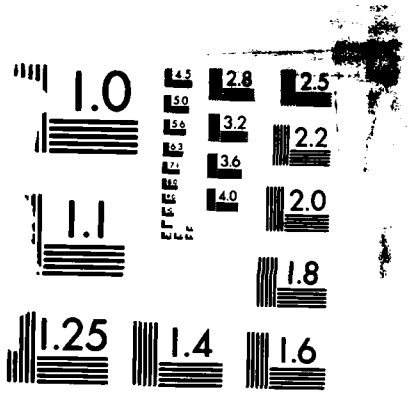
AD-A165 937 BASIC INSTRUCTIONAL PROGRAM: SYSTEM DOCUMENTATION(U) 1/1
STANFORD UNIV CA INST FOR MATHEMATICAL STUDIES IN THE
SOCIAL SCIENCES M L DAGEFORDE MAY 78 NPRDC-TN-78-12
UNCLASSIFIED N00123-76-C-1543 F/G 9/2 NL





NAVY PERSONNEL RESEARCH AND DEVELOPMENT CENTER SAN DIEGO, CALIFORNIA 92152

NPRDC TN 78-12

MAY 1978

BASIC INSTRUCTIONAL PROGRAM: SYSTEM DOCUMENTATION

Technical Note 78-12 May 1978

BASIC INSTRUCTIONAL PROGRAM: SYSTEM DOCUMENTATION

Mary L. Dageforde

Institute for Mathematical Studies in the Social Sciences
Stanford University
Stanford, California 94305

Reviewed by
John D. Ford, Jr.

Navy Personnel Research and Development Center
San Diego, California 92152

FOREWORD

This research and development was conducted in response to Navy Decision Coordinating Paper, Education and Training Development (NDCP-Z0108-PN) under subproject Z0108-PN.32, Advanced Computer-Based Systems for Instructional Dialogues, and the sponsorship of the Director, Naval Education and Training (OP-99). The overall objective of the subproject is to develop and evaluate advanced techniques of individualized instruction.

This is one in a series of six reports dealing with the BASIC (Beginner's All-Purpose Symbolic Instruction Code) Instructional Program (BIP), which is a "tutorial" programming laboratory designed for the student who has had no previous training in programming.

Previous reports in the program are concerned with conversion of BIP into the MAINSAIL programming language (Note 1, 1978), the BIP supervisory-level manual (Note 2, 1978), BIP student manuals (Notes 3 and 4, 1978), and curriculum information networks for computer-assisted instruction (Beard, Barr, Gould, & Wescourt, 1978). This report is intended for use by individuals involved with the system-level support of the BIP system.

The work was performed under Contract N00123-76-C-1543 to Stanford University. The contract monitors were Dr. John D. Fletcher and Dr. James D. Hollan.

J. J. CLARKIN
Commanding Officer

SUMMARY

The BASIC Instructional Program (BIP) is a "hands-on laboratory" that teaches elementary programming in the BASIC language. This report documents the BIP system as implemented in MAINSAIL. MAINSAIL is a machine-independent revision of SAIL which should facilitate implementation of BIP on other computing systems. Each of the modules which make up the system is described in detail.

CONTENTS

LIST OF FIGURES

SECTION 1. INTRODUCTION

The BASIC Instructional Program (BIP) is an interactive problem-solving laboratory that teaches elementary programming in the BASIC language. It was developed on the IMSSS PDP-10 research computer facility in a specialized high-level language called SAIL (Reiser, 1976), which is presently available only on PDP-10 computers. During the year starting in October 1976, BIP was rewritten (Dageforde, Note 1) in the programming language called MAINSAIL (MAchine-INdependent SAIL) (Wilcox, 1977a) being developed at the Stanford University Medical Experimental (SUMEX) Computer Facility. MAINSAIL, as reflected in its name, provides capabilities similar to those in SAIL independently of the underlying computer system (Wilcox, 1977b). It is designed to be powerful and efficient, with a high degree of portability on a broad class of computers. Thus, BIP was rewritten in MAINSAIL so that implementation on other (notably smaller) systems would be possible.

BIP was written in eight separately-compiled models (ONOFF, BIP, PARSE, XERS, ERRDOK, TEACHR, VERFY, and MSRECS) that are brought into memory (by the MAINSAIL runtime system) during execution as needed. The following sections describe the workings of all those modules as well as the curriculum data structures and the information saved in student histories.

SECTION 2. ONOFF MODULE

The ONOFF module performs three major tasks at student sign-on or sign-off:

1. Checks that the student is enrolled in the course.
2. Sets up the curriculum data structure.
3. Reads from and writes to the student's history file.

2.1 Checking that Student is Enrolled in Course

The text file WHO contains each BIP student's number and name. When a student signs on by typing his number and first name, BIP searches the WHO file for a line with that information. If no such number is found, or if the name typed does not match the name in the appropriate line, BIP tells the student that the number and/or name are incorrect, and logs him off.

2.2 Setting Up the Curriculum Data Structure

The curriculum for BIP is contained in a text file called TASKS. Before any students run BIP, the TODATA program is run to compress certain essential information from TASKS and write it onto a data file called INIT. When a student signs on, the INIT data is used to initialize the curriculum data structure. Throughout a student's session, BIP reads from the TASKS file to access the text of the current task, its hints and model, etc. The pointers that were initialized from the INIT data give BIP efficient access to the text in the TASKS file. There are two data structures that need to be initialized: one for the techniques, and one for the tasks.

The pointer array technique has an element for each of the 16 programming techniques in the curriculum (see Barr, Beard, & Atkinson (1976), for a description of the curriculum structure). Each element points to the start of a linked list of the numbers of the skills found in that technique. (See the appendix for a list of the techniques and the skills within those techniques.) This information about the sets of skills in each technique is used by the task-selection algorithm (see Section 7). As shown in Figure 1, the first technique includes skills 1, 2, 5, and 8; the second, skills 3, 4, 6, 7, 9, 10, 11, and 12, etc.

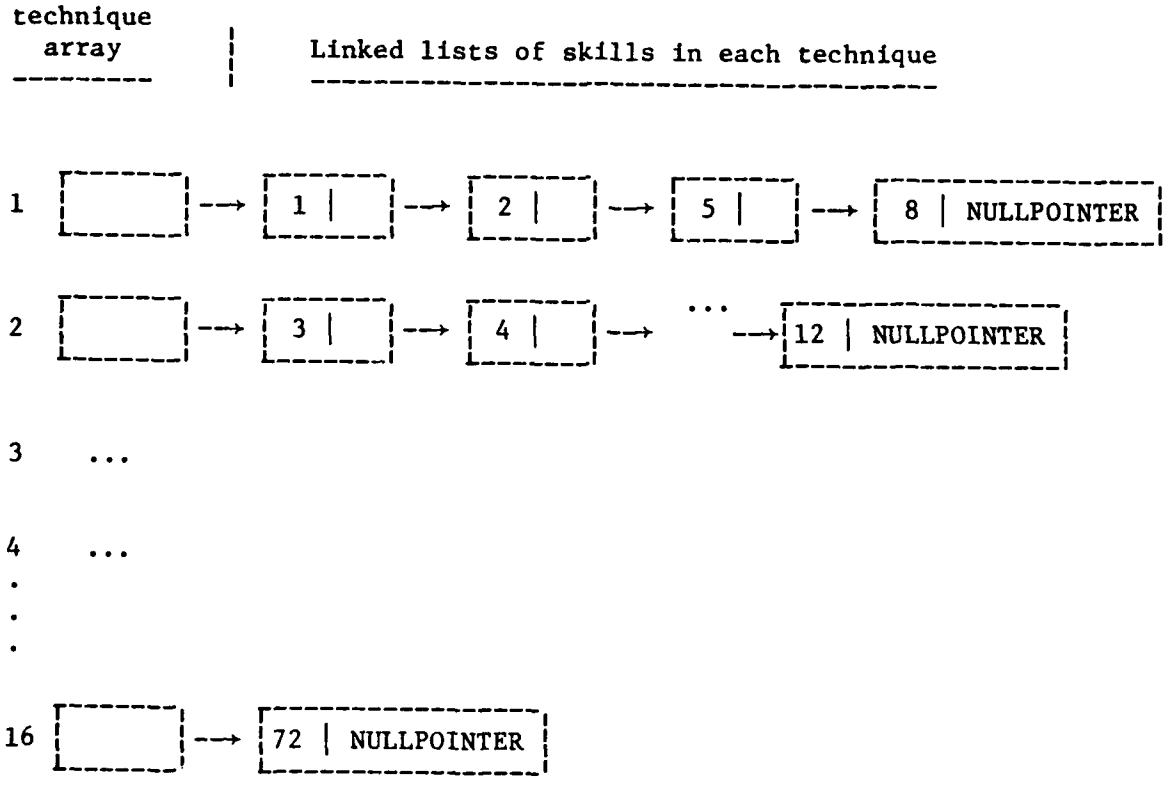


Figure 1. Data structures for lists of skills in techniques.

The data structure for the tasks is a linked list of records, one for each task. Each record has 13 fields:

1. "link," a pointer to the next main task in the linked list.

2. "name," the name of the task.

3. "taskIndex," the task number.

4. "taskPos," the start position (in the TASKS file) of the task description.

5. "modelPos," the start position (in the TASKS file) of the model solution.

6. "firstHintPos," the start position (in the TASKS file) of the first hint. If there are not any hints, it is set to zero.

7. "nextHintPos," initially, the start position (in the TASKS file) of the first hint (if any). If there are no hints, it is set to zero. During a BIP session, after the student has seen a hint, nextHintPos is set to the start position of the next hint, if it exists, and to zero otherwise.

"moreTask," a pointer to the start of a linked list of the "moreTasks," or extensions, of this task (if none exist, it is set to NULLPOINTER). Each moreTask record is exactly like a main task record, except that the "link" field is not used, and the "moreTask" field points to the next record in the linked list of moreTasks of the current main task.

9. "reqOps," which has a bit turned on for each BASIC operator required in the student's solution to this task (see below).

10. "disOps," which has a bit turned on for each BASIC operator that the student may not use in the solution to this task (see below).

11. "reqFns," which has a bit turned on for each function (INT, RND, and/or SQR) required in the student's solution to this task (see below).

12. "disFns," which has a bit turned on for each function that the student may not use in the solution to this task (see below).

13. "skills," a pointer to a linked list of the programming skills used in the solution to this task.

Each "reqOps" and "disOps" field consists of 16 bits, one for each BASIC operator that can be required or disabled in a student's program:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	INPUT		IF		DIM		FOR		GOSUB		READ		REOPEN		ENSUB
LET		GOTO		REM		STOP		NEXT		RETURN		DATA		BEGINSUB	

Thus, if LET, GOTO, and STOP were required in the solution to a particular task, the reqOps bits for that task would be 1010001000000000, where a 1 indicates that the bit is turned on and a 0 that it is turned off.

Each "reqFns" and "disFns" field uses 3 bits, one for each function that could be required or disabled in a student's program:

2	1	0
INT	RND	SQR

Thus, if INT were disabled in the solution to a particular task, the disFns bits for that task would be 100.

2.3 Reading From and Writing to the History File

Each individual student has a personal history file, a data file used to store information about the student's current state (what task he is currently working on, how many tasks completed so far, etc.), and past performance on tasks and skills. At sign-on, this file's information is read into variables and arrays whose elements are modified during the BIP session. At sign-off, and at numerous other points during the session, the updated information is written back out to the history file.

The history files contain the following information:

1. The History Summary

integer variables

variables	information
studentNum	student number
lastDateSignon	date of last session
lastTimeSignon	time of last session
totTimeOn	total time on (minutes)
totNumSessions	total number of sessions
numTsksDone	total number of tasks completed
lastTnqUsed	highest technique used in last task selection
numTnqToStayIn	0 - if no technique to "stay" in n - if task-selection algorithm should stay at technique n, since student had trouble with last task at that level
chronIndex	Index into chron array (see below)
mainNum	number of last main task seen
taskNum	number of last task seen (either the same as mainNum or the number of a moreTask of the last main task seen)

bits variable

variable	information
tnqWord	16 bits, one for each technique (bit 0 for tnq 1, 1 for tnq 2, ..., 15 for tnq 16). A bit is off if corresponding technique has never been seen, on if it has.

2. The Task History--BITS array tskInf. Two bits-elements per task. (BITS is a MAINSAIL data type for representing a short sequence of bits.)

1st BITS	contents	changed in routine
0	never see task again	upVer, enufo, verOption
1	ever passed verifier	upVer (called by moreo)
2	passed on first try?	upVer
3	ever said he understood task	postTaskInt
4-8	# of failures in verifier	moreo
9	chose to leave after failure?	verOption (called by verfy)
10	disagreed with verifier?	verOption
11	saw the model	modelo
12	saw all the hints	hinto
13-15	number of hint requests	hinto

2nd BITS		
0-2	free	
3-7	times task was seen	finTask (called by moreo, enufo)
8-15	minutes on task	addTime (teacho, moreo, enufo)

3. The Skill History--BITS array sklInf. Three bits-elements per skill.

1st BITS	contents	changed in routine
0-5	times seen	finTask
6-10	times task passed verifier	upVer (called by moreo)
11-15	times passed verifier in a row	upVer & downVer

2nd BITS		
0-5	free	
6-10	skill "ok" in post-task int.	postTaskInt
11-15	skill "ok" in a row	postTaskInt

3rd BITS		
0-7	free	
8-15	minutes on skill	addTime

4. The Chronological Order of Tasks--BITS array chron. Two bits-elements per task, in the order the student completed the tasks.

1st BITS	contents	changed in routine
0	on if task was specifically requested by student	teacho
1	ever passed verifier	upVer
2	passed on first try	upVer
3	on if "understood" task	postTaskInt
4-8	number of failures in verifier	moreo
9	chose to leave after failure	verOption
10	disagreed with verifier	verOption
11	saw the model	modelo
12	saw all the hints	hinto
13-15	number of hint requests	hinto

2nd BITS		
0-7	task identification number	
8-15	minutes on task	addTime

5. The Data and Time Each Task was Started--integer array chronDaTime. There are two elements per task, the first telling the date, and the second, the time of the start of the task, referenced by the corresponding elements of the chron array.

6. The Student-assigned File Names--string array stuFile. The ASCII character codes of the characters in the student-assigned file names are saved at the end of the student history, with the different names separated by 32, the character code for a space.

SECTION 3. BIP MODULE

The BIP module contains the main BIP program procedure execlin, which scans each line typed by the student to decide whether it is a BIP command, a BASIC statement, a request for help, or an error. Execlin then calls an appropriate procedure to either follow the command, to parse the statement, or to print a help or error message.

The BIP file's other main procedures handle student files, execute programs, print help messages when requested by the student, and save protocols of the BIP session (when desired by the supervisor).

3.1 Execlin Procedure

When a student signs on, the BIP program performs some initialization and preparation for the session. Then it repeatedly calls the main procedure execlin to handle each input line, calling appropriate procedures depending on whether the line is a BASIC statement, a BIP command, an error, or a request for help.

If the first character of the line is a "?," the help procedure (described below) is called.

If the first character is a number, execlin expects the line to be a BASIC statement. It scans over the line number and expects the next sequence of nonblank characters to be a BASIC operator (LET, INPUT, PRINT, etc.). If it is a BASIC operator, then the procedure syntax is called to parse the statement. If it is not, execlin checks to see if the line is a user-function definition or a LET statement without explicit use of the word "LET," and if so, calls syntax. Otherwise, the line has an error: either it is missing a valid BASIC statement, or it contains a BIP command following a line number.

If the input line does not start with a number, execlin expects it to be a BIP command. If it is, the appropriate procedure to carry out that command is called. Otherwise, the line has an error: either it is an illegal command or it has a BASIC operator which is missing a line number.

If any of the above-mentioned possible errors occurs, execlin calls the procedure msgTxt to get the appropriate error message, prints that message, tells the student that the input line was not accepted, and indicates that he may type "?" for help.

3.2 Dealing with Student Files

Students are allowed to save up to 10 of their programs for later use. At any point, they can save the current program under a name that they assign. Later they can retrieve that program with the -GET- command, or delete it with the -KILL- command. At any point, the -FILES- command will list the names of the files currently stored.

The name students think their program is saved under and the name it is actually stored under are two different things. It is necessary to

assign the programs unique names so that they are not confused with other programs saved under the same names by other students. Also, in order for a file name to be truely machine-independent, it should consist of no more than six characters. Therefore, the format of the name under which the program is actually stored is

S<student number>F<file number>

where <file number> is a number between 0 and 9.

During a BIP session, the student file names are in the string array stuFile. If the student's number were, say, 88, then stuFile[0] would be the student-assigned name for the file S88F0, stuFile[1] would be the name for S88F1, etc. Between sessions, the student file names are stored at the end of the student history.

When a student asks to save a program, the procedure saveo searches the stuFile array for the next null element, assigns it the name the student gave his program, creates a new file whose name is in the format described above, and copies the student program to that file.

To retrieve a file, geto searches stuFile for the appropriate student file name (and tells the student if that name does not exist), opens the corresponding file, and copies the saved program into the student's working space.

To delete a file, killo searches stuFile for the appropriate student file name, deletes the corresponding file, and then sets that element of stuFile to null, so that it can be subsequently used when the student wants to save another program.

To list the saved student files, fileso simply steps through the stuFile array and types out the nonnull elements.

3.3 Executing Programs

The procedure runo is called to execute a program. First it calls doktor (the main procedure in the ERRDOK module--see Section 6) to check the program for structural errors. If none exist, it commences interpretation of the program. For each statement, runo sends xcute, the main XERS procedure, the BASIC operator used in the statement, and then xcute calls the appropriate procedure to interpret it. Runo keeps track of the number of statements that have been executed and warns the student when it is excessive, since the program may be in an infinite loop. At that point, the student has the option of either stopping execution or continuing and telling runo the maximum number of further statements to execute.

3.4 The Help System and the MSGS File

At any point, the student may type a "?" for help. The procedure help is called to decide what type of help the student needs. If the "?"

is typed immediately following a syntax, structural, or execution error, help will give the student further information about the type of error made and tell him to type "?" for more help (as long as it is available--there are up to four different help messages available for each error). If the student types "?REF," help refers him to a section of the Student Manual.

If the student types "?" at any other time (i.e., not after making an error), help will simply state that BIP is expecting either a BIP command or a BASIC statement and that he can type "?BASIC" to see a list of the statements and commands.

How does help know which help message to give at any point after an error has been made? First of all, it knows which type of error was made, since exactly one of the variables synerf, dokflg, or xerflg (for a syntax, structural, or execution error, respectively) will be nonzero and will tell the error number. Help keeps track of how many "?"s have been typed directly after an error so that it knows which message (in the sequence of different help messages for the error) to print out.

The help messages are all in the file MSGS, along with the error messages, the manual reference messages, and the skill descriptions. Each group of messages is on a separate page of MSGS, in order. The beginning of each page has pointers to the start of all the messages on that page. Given a page number and a message number, the procedure msgTxt will retrieve the appropriate message from the file MSGS. Help knows the page number for each type (syntactic, structural, or execution) of error made, and the error number from synerf, dokflg, or xerflg. So it simply computes which message number should be presented (based on the number of times the student has typed "?") and calls msgTxt with that number and the appropriate page number to get that message.

3.5 Protocol Saving

The Supervisor at each BIP implementation has the choice of whether or not to compile the code for protocol saving. This is done by setting the macro "canSaveProtocol" (in the file MACROS) to TRUE or FALSE just before compiling the BIP modules. If it is TRUE, then the Supervisor may save "protocols," or records of all that happens during student sessions, for some or all BIP students. Whether or not they will in fact be saved for individual students is determined during the creation of student histories, by an option in the newHst program.

The protocol-saving code is scattered throughout the BIP modules. The BIP file contains two procedures that are often called by that code to write various information to the protocol file. One, writeTasks, writes the name and number of the task the student is currently working on. The other, writeProg, writes the current student program.

If the protocol-saving code was compiled and the Supervisor said he wanted protocols saved for a specific student, then a protocol of each of that student's BIP sessions will be saved on his personal protocol file, whose name is of the format DAT<student number>.

SECTION 4. PARSE MODULE

The purpose of the procedures in the PARSE module is to examine each line of BASIC code that the student types and to produce a line of internal code that can be read by the procedures in the XERS module.

4.1 Recursive Control

The parser uses a very common method called top-down parsing with recursive descent to scan the input line and to produce an argument line for XERS (the interpreter). The best way to explain this method is with an example. Suppose the statement to be parsed is a LET statement, which has the following syntax:

<variable> = <expression>.

In this case, the parser first looks for a legal BASIC variable. If it finds one, then it examines the input string, expecting an "=." If it does not find it, an error has obviously occurred. Otherwise, the parser goes ahead and looks for an expression.

Thus, the first action of the parser is to call the procedure variableParse. In the same way that lets (the procedure called by syntax, the parser control procedure, to parse a LET statement) "knows" the legal syntax of a LET statement, variableParse "knows" what the correct syntax of a variable is:

<variable> = <string variable> or <numeric variable>

<string variable> = <string id> or <string id> (<arith.exp>)

<numeric variable> = <numeric id> or <numeric id> (<arith.exp>)
or <numeric id> (<arith.exp>, <arith.exp>)

Hence, it checks the first part of the line to see whether the variable is string or numeric. In either case, the parser then looks for a "(," because the variable might be an array element. If no "(" is found, control returns to lets, because the parser assumes the variable has been found. If a "(" is found, however, variableParse passes control to aExp, the arithmetic expression parser. Upon return from aExp (in the case of a string variable), variableParse looks for the closing parenthesis. If one is found, control is passed back to lets. If not, then a syntax error has occurred--either a parenthesis mismatch or an illegal arithmetic expression.

The process continues in this manner. AExp immediately passes control to aTerm which passes control to aFactor, because of the syntax of arithmetic expressions:

<arith.exp> = <arith.term> or <arith.term> + <arith.term>

<arith.term> = <arith.factor> or <arith.factor> * <arith.factor>

<arith.factor> = <arith.primary> or <arith.primary> ^ <arith.primary>.

Finally, in aPrimary the parser looks for an arithmetic primary such as a numeric constant, a numeric variable, or a user-defined function. If it finds a legal primary, it returns control back to aFactor. Otherwise, a syntax error must have occurred.

Eventually, control will be passed back up the line to lets (unless a syntax error occurs), which will then look for the "=" symbol, call the expression parser, and return back to syntax. For example, the statement

Z (X + 2) = INT (Y) would be parsed as follows:

<LET statement>						
<variable>					=	<expression>
<num var>	(	aExp		)	=	aExp
Z	(aTerm	+	aTerm)		=	aTerm
Z	(aFactor	+	aFactor)		=	aFactor
Z	(aPrimary	+	aPrimary)		=	aPrimary
Z	(X	+	2	)	=	INT (aExp)
Z	(X	+	2	)	=	INT (aTerm)
Z	(X	+	2	)	=	INT (aFactor)
Z	(X	+	2	)	=	INT (aPrimary)
Z	(X	+	2	)	=	INT (Y)

This indicates exactly which procedures would be called in the parsing of the statement. Each procedure calls the one beneath it in the hierarchy and returns control back to the procedure which called it if it finds what it expects. (Thus, the name "recursive descent" for this parse method, because the parser descends through levels of procedures until it finds a match for the part of the statement it is looking at.)

4.2 Types and Tokens

However, the parser must know exactly how much of the line is to be examined at any given time. To do this, it uses a procedure called getToken, which takes the next syntactically meaningful part of the input string and puts it in the variable token. At the same time, it "types" this token--that is, it assigns to it a number that indicates what its meaning is. These types are defined as macros in the file MACROS (e.g., inteqv is defined as 17) so that they can be used in CASE statements. The use of mnemonic macro names instead of integers makes the code more readable. The macro names for the types are listed on the following page.

nid	numeric variable
sid	string variable
strcon	string constant
numcon	numeric constant
oper	+-*/&() ;
function	a user-defined function
eqal	an = or _
noteq	<>
less	<
leseq	<=
greq	>=
greater	>
nott	boolean not
andd	boolean and
orr	boolean or
root	square root
inteqv	truncated integer
rndnum	random number
lengt	length of a string
bad	illegal character
narray	one-dimension numeric array
dnarray	two-dimensional numeric array
sarray	string array

For example, if the input line were Z (X + 2) = INT (Y), the procedure getToken would pass the line to the other procedures of the parser as follows:

```
nid Z, oper (, nid X, oper +, numcon 2 oper ), eqal =,
inteqv INT, oper (, nid Y, oper ).
```

(Later, during the variable-parsing routine, the variable Z is recognized as an array variable, so its type would be changed to narray for the interpreter.)

Hence, each time getToken is called, it gets a new "meaningful syntactical entity" from the input string, types this "entity," and places the values into the variables token and type. GetToken first gets the next nonblank character on the string. Then, depending on what the character is, getToken assigns it a "preliminary type" obtained directly from the array typetable (initialized in the module Pcom, which is listed in the file PARSE), which has an entry for every ASCII character (e.g., typetable[+] = oper, typetable[X] = nid, typetable[!] = bad).

GetToken then uses this preliminary type to decide what to do next. For example, if the character is a letter, it scans the input line until a nonalphanumeric character is encountered, and passes the resulting string to another procedure typeId, which determines if the correct type is a numeric variable, string variable, user function, etc. If a " is encountered, then getToken scans the line for another ", and assumes that whatever lies between the two is a string constant. If the character is a +, /, *, or -, nothing is done.

The parser then uses the information passed to it by getToken to decide what to do next. For example, in the procedure aPrimary, if the current type is inteqv, the parser assumes (perhaps erroneously) that a value for the INT function is forthcoming. Hence, it calls getToken to be "(" after the call. Then it calls get Token again, and then aExp, because it assumes the argument for the INT call will be an arithmetic expression. Upon return from aExp, it expects that token will be ")"—if not, a syntax error has occurred.

The main problem with this type of parsing algorithm occurs when it is not immediately obvious what to do next even knowing what the current token is. In BIP's parser, this situation occurs while trying to parse general expressions (in PRINT statements) and Boolean expressions. For example, suppose we are attempting to parse an expression in a PRINT statement, and the current token is X. Although this eliminates a string expression from consideration, the parser does not know yet whether the expression is arithmetic or Boolean; the statement could be PRINT X or PRINT X=Y. Normally parsers try to solve this problem with a method called backup, perhaps trying to parse an arithmetic expression first. If this fails, the input string can be restored to its original state, and a Boolean parse tried.

BIP's parser uses methods that try to determine the type of the expression without parsing it. For example, to determine whether the expression is Boolean, the line is scanned, breaking on "=" or ">" or "<"—which must be present in a Boolean expression. If one is found, the expression is assumed to be Boolean; otherwise, it must be arithmetic or string. In either case, the line is restored to its state before the scan, and the parse proceeds correctly.

The same type of problem can occur while parsing a Boolean expression, where the parser expects to find a string or arithmetic expression followed by a Boolean symbol and then another arithmetic or string expression. If the token is a "(" or a user-defined function, the parser has no way of telling at that point whether the expression is string or arithmetic. Again, the input string is scanned for the type of expression it contains.

4.3 Polish Notation

In the process of parsing the line, the parser transforms it from infix (normal parenthesizing) to polish notation, in which operators follow their arguments. For example, the expression

```
INT (SQR (X * (2 + Y)))
```

becomes

```
X 2 Y + * SQR INT
```

in the polish notation. Also standard delimiters are used to signal the beginning of an array subscript and the end of an expression. For example,

A (3, Y + 2) becomes A (3, 2 Y +),

[(used to denote beginning of an array or substring]
[, used to separate subscripts or string characters]
[) used to denote the end of an array or substring]

For the separation of expressions in a PRINT statement, the nonprinting character whose character code is 30 is used. For the separation of variables in an INPUT, DIM, or READ, and the separation of data, a ";" is used. As an example,

INPUT A, A$, B$ (J) becomes A; A$; B$ (J).

The way the parser transforms the line from infix to polish notation is rather straightforward. In the "INT (SQR (X * (2 + Y)))" example above, while parsing the INT," instead of generating the code for the "INT" before aExp is called, the parser waits until after the return from aExp. Hence, the code for everything else has already been produced and the "INT" is tacked on the end. The same thing is done whenever any operator is encountered.

Thus, the code for "SQR" is tacked onto the end of X 2 Y + *. In the case of a binary operator (one that has two arguments, such as + or *), basically the same thing is done. The code for the X is produced in aPrimary, and then control passes back to aTerm. Instead of making the code for the * at this stage, aTerm calls aPrimary a second time, which then calls aExp to parse the parenthesized expression. AExp calls aPrimary which adds the code for the 2, but when control returns to aExp, it does not generate the code for the +, but waits until after the second call to aPrimary (which adds the code for the Y). Hence, at this point the code is "X 2 Y +." Finally, control passes back to aPrimary, which finds the end of the parenthesized expression, and then to aTerm, which finally generates the code for the *. (What actually happens is probably less confusing than this description.)

4.4 Format of the Code Produced

The code has a special format (accumulated in the string variable kode and then stored in an element of the argz array) so that it can be easily scanned by XERS. An element of code has the format

<type><token><delimiter>

and elements of code are strung together to form the complete expression. Each complete line of internal code, corresponding to a BASIC expression typed by the student, it stored in an element of the argz array, which is interpreted a line at a time, by the procedures in the XERS module.

Nonprinting characters whose ASCII codes are 1 through 23 are used for the <type> and the nonprinting character 29 is used for the <deliminter>

in the above format specification. Using lower-case letters (a = 1, b = 2, etc.) to indicate the <type> and the] character to indicate <delimiter> the internal code produced by the parser for the example discussed above is:

```
a X ] b 2 ] a Y ] e + ] e * ] p SQR ] q INT ]  .
```

4.5 Arrays Affected by the Parser

Finally, the parser updates five arrays (prgtxt, linNums, opers, argz, and order) each time a BASIC statement is parsed. Prgtxt holds the exact text that was typed; linNums, the input line numbers; opers, the BASIC operators; and argz, the code (as produced by the parser) for the line. Every time a new (syntactically correct) line is typed by the student, the next element of each of those four arrays is assigned (in the procedure putLin) appropriate values from the information in that line. The integer array order tells the numerical (by line number) order of the lines. The value of the first element of order is the index into the other four arrays of the line with the lowest line number; the value of the second element is the index of the line with the next higher number, etc.

For example, if the student input the following lines as a solution to the simple task "SPOON""

```
order typed                    numerical order
-----------                    ---------------
 10 READ X$                     10 READ X$
 20 READ Y$                     20 READ Y$
100 DATA "SILVER", "SPOON"      30 PRINT X$ & " " & Y$
 30 PRINT X$ & " " & Y$        100 DATA "SILVER", "SPOON"
999 END                        999 END
```

then the values of the prgTxt, linNums, opers, and order array elements are:

index	prgTxt	linNums	opers	order
1	10 READ X$	10	READ	1
2	20 READ Y$	20	READ	2
3	100 DATA "SILVER","SPOON"	100	DATA	4
4	30 PRINT X$ & " " & Y$	30	PRINT	3
5	999 END	999	END	5

SECTION 5. XERS MODULE

The workings of the interpreter are relatively simple when compared with the parser. Because the argument code generated by the parser (and stored in the array argz) is in polish notation, it is simple to use stacks to evaluate expressions. Again, an example is appropriate. Suppose we wish to evaluate the polish expression

X 2 Y + * SQR INT.

The heart of XERS, the procedure evalToken, is designed to perform certain actions depending on what type the current token in the args string is. (To get another token off the args string, the procedure nextx is called. It scans to the next] delimiter, putting a new token and type in the variables of the same name.) In this example, evalToken would do the following as each token is scanned:

X	Push the value of X onto the real stack.
2	Push 2 onto the real stack.
Y	Push the value of Y onto the real stack.
+	Pop the top two values off the real stack, add them, and push the result onto the real stack.
*	Pop the top two values off the real stack, multiply them and push the result.
SQR	Pop the top value off the real stack, take its square root, and push the result onto the stack.
INT	Pop the top value off the real stack, truncate it, and push the result back onto the real stack.

Hence,

INT (SQR (X * (2 + Y)))

has been evaluated.

Any expression is evaluated in the same way. A slight complication occurs if we wish to evaluate the value of an array variable. In this case, we make use of a procedure called goUntil, which calls the token evaluator (evalToken) until a specified delimiter character is reached. For example, suppose we wish to evaluate the value of

A$ (X + 2) & X$, in polish form: A$ (X 2 +) X$ &.

Upon getting A$, since the interpreter realizes it is an array variable, it does not try to push its value onto the stack. Rather, it scans past the "(" symbol, and then calls goUntil to call the token evaluator until a ")" is reached. Hence, the following occurs:

X	Push the value of X onto the real stack.
2	Push 2 onto the real stack.
+	Pop the top values off, add them, push the result.
)	Stop.

At this point, the value of X + 2 is sitting on top of the stack. So the interpreter pops its value off the real stack, pushes the value of A$ (X + 2) onto the string stack, and then does the following:

X$	Push the value of X$ onto the string stack.
&	Pop the top two values off the string stack, concatenate them, and push the result back onto the string stack.

Assignments are made in a similar manner. One important thing should be noted--the interpreter assumes that the value we wish to assign to the variable is the top element of the stack just before the assignment is to be made.

Suppose, for example, that we want to make the following assignment:

A = B (J) + INT (Y) which has the following polish form:

B (J) Y INT + = A

In this case, the interpreter's first instruction is goUntil("="), which causes the evaluator to be called until a "=" is reached. At this point, the value of B (J) + INT (Y) will be on top of the stack. Upon examining the A, the evaluator will pop the stack, and store that value in the variable A.

SECTION 6. ERRDOK MODULE

When a student gives the RUN command to BIP, the ERRDOK module checks the student's program for a number of possible structural errors.

ERRDOK employs a two-pass algorithm. If, at any point, an error is encountered, an appropriate error message is given and the program is not executed.

ERRDOK initially makes sure that there exists a program to be run in the first place, and that the last line is an END statement. If so, it then checks the program line by line for further possible errors.

There are a few key variables and record lists employed by ERRDOK in its first pass through the lines of the program.

1. inSub is a BOOLEAN that is TRUE whenever a BEGINSUB has not yet been followed by an ENDSUB.

2. loopLevel tells the number of FOR statements that have not yet been followed by NEXTs.

3. forVar is a STRING array with the names of the FOR loop index variables.

4. usersFns is the record class for a linked list of records, one for each function defined in the program. It has three fields: "name," for the function's name; "index," telling the location of the function definition in the program; and a link to the next record.

5. SUBclass is the record class for a linked list of records, one for each subroutine in the program. It has four fields: "subStart," for the BEGIN line number; "subEnd," for the ENDSUB line number; a BOOLEAN "referenced," set only when a GOSUB references the subroutine; and a link to the next record.

6. FORclass is the record class for a linked list of records, one for each FOR loop in the program. It has three fields: "forStart," for the FOR line number; "forEnd," for the NEXT line number; and a link to the next record.

If the line being checked in the first pass gives a function definition, ERRDOK checks the already existing userFns records (if any). If the "name" field of any of them is the same as the name of the function being defined, then the function has been defined twice, and an error message is given. Otherwise, a new userFns record is created.

An END statement that is not the last line of the program produces an "Illegally located 'END'" error message.

If the line being checked is a FOR statement, the loopLevel number is incremented and the array element forVar[loopLevel] is set to the FOR loop's index variable. A new FORclass record, with the "forStart" field set to the current line, is created.

When a NEXT statement is encountered, loopLevel is checked. If it is zero, then the program has a "'NEXT' without a preceding 'FOR'" error. Otherwise, forVar[loopLevel] is checked. If it is not the same as the variable in the NEXT statement, an "Illegally nested FOR...NEXT loop" error has been detected. Otherwise, the "forEnd" field of the FORclass record for the loop just ending is set to the NEXT statement's line number and loopLevel is decremented.

If the line being checked is a BEGINSUB statement and the BOOLEAN variable inSub is TRUE, then an "Illegally imbedded subroutine" error has been found--a subroutine has begun inside another subroutine. Otherwise, inSub is set to TRUE, and the previous non-REM statement is checked. If it is not a STOP, a GOTO, or an ENDSUB statement, then there is an error: execution of the student's program could illegally fall through into the subroutine. Otherwise a new SUBclass record, with the "subStart" field set to the current line number and the "referenced" field initialized to FALSE, is created.

When an ENDSUB statement is encountered, if inSub is not TRUE, then the program has an "'ENDSUB' without a preceding 'BEGINSUB'" error. Otherwise, the "subEnd" field of the SUBclass record for the subroutine just ending is set to the ENDSUB statement's line number, and inSub is set to FALSE.

Other statements are ignored during the first pass.

If the first pass is complete and the variable inSub is TRUE, then a "Missing 'ENDSUB' after 'BEGINSUB'" error has occurred. And if loopLevel is not zero, then a "'FOR' statement without matching 'NEXT'" error has occurred.

Otherwise, everything is all right so far, and all GOSUB, GOTO, and IF statements are checked in the second pass.

GOSUB statements are checked to make sure that they branch only to BEGINSUBs. And if the GOSUB is located within the subroutine branched to, an error has occurred, since recursive subroutines are not allowed. (This error has occurred if the GOSUB's line number is between the called subroutine's beginning and ending line numbers--its SUBclass record's "subStart" and "subEnd" values.) If all is well, the "referenced" field of the subroutine called by the GOSUB is set to TRUE.

GOTO and IF statements are checked to make sure that the line to which they branch exists and is an executable statement (any statement other than a DATA or a DIM). Then an illegal branch into the middle of a FOR loop or into or out of a subroutine is checked for.

Once the second pass is complete, the "referenced" field of each SUBclass record is checked. If it is not TRUE, then an error has occurred, since all subroutines must be referenced by at least one GOSUB.

Finally, if no structural errors have been detected by ERRDOK, the program is allowed to run.

SECTION 7. TEACHR MODULE

TEACHR's main procedures handle task selection, the post-task interview, updating of the student history, and stepping the students through their first session.

In order to understand how task selection is done, it is necessary to understand the definition and use of skills and techniques. BIP's curriculum goals are the mastery of certain programming techniques, including simple output; using loops, conditional branches, and arrays; assignment to variables, etc. The techniques are linked in a linear order, each having but one "prerequisite" (the previous technique), based on dependence and increasing program complexity.

The techniques are interpreted as sets of skills, which are very specific curriculum elements like "printing a literal string" or "using a counter variable in a loop." The skills are not themselves hierarchically ordered. The appendix lists the techniques and skills within them. The programming problems or "tasks" are described in terms of the skills they use, and are selected on the basis of this description, relative to the student's history of competence on each skill. Figure 2 shows a simplified portion of the curriculum network, and demonstrates the relationship among the tasks, skills, and techniques.

The algorithm by which BIP selects a next task when the student requests it is shown in Figure 3. The selection process begins with the lowest (least complex) technique. The procedure setUpSets puts all the skills in that technique into a "set" (actually, a linked list of skill records) called MAY, which will become the set of skills that the next task "may" use.

SetUpSets then examines the student's history on each of the skills associated with the technique, to see if it needs further work. Two key counters in the history (see documentation for ONOFF) are associated with each skill. One is based on the results of the solution checker, and monitors the student's continuing success in using the skill. The other is based on the student's self-evaluation, and monitors his own continuing confidence in the skill. The current definition of a "needs work" skill is one on which either counter is zero. For each successful use of a skill, both counters are incremented (in upVer). If the student quits a task requiring a particular skill, the first counter is decremented; if the student requests more work on a skill (during the post-task interview, described below), the second counter is zeroed. Any such "not yet mastered" skills are put into the MUST "set" (linked list of skill records). Eventually the program will seek to find a task that uses some of these MUST skills.

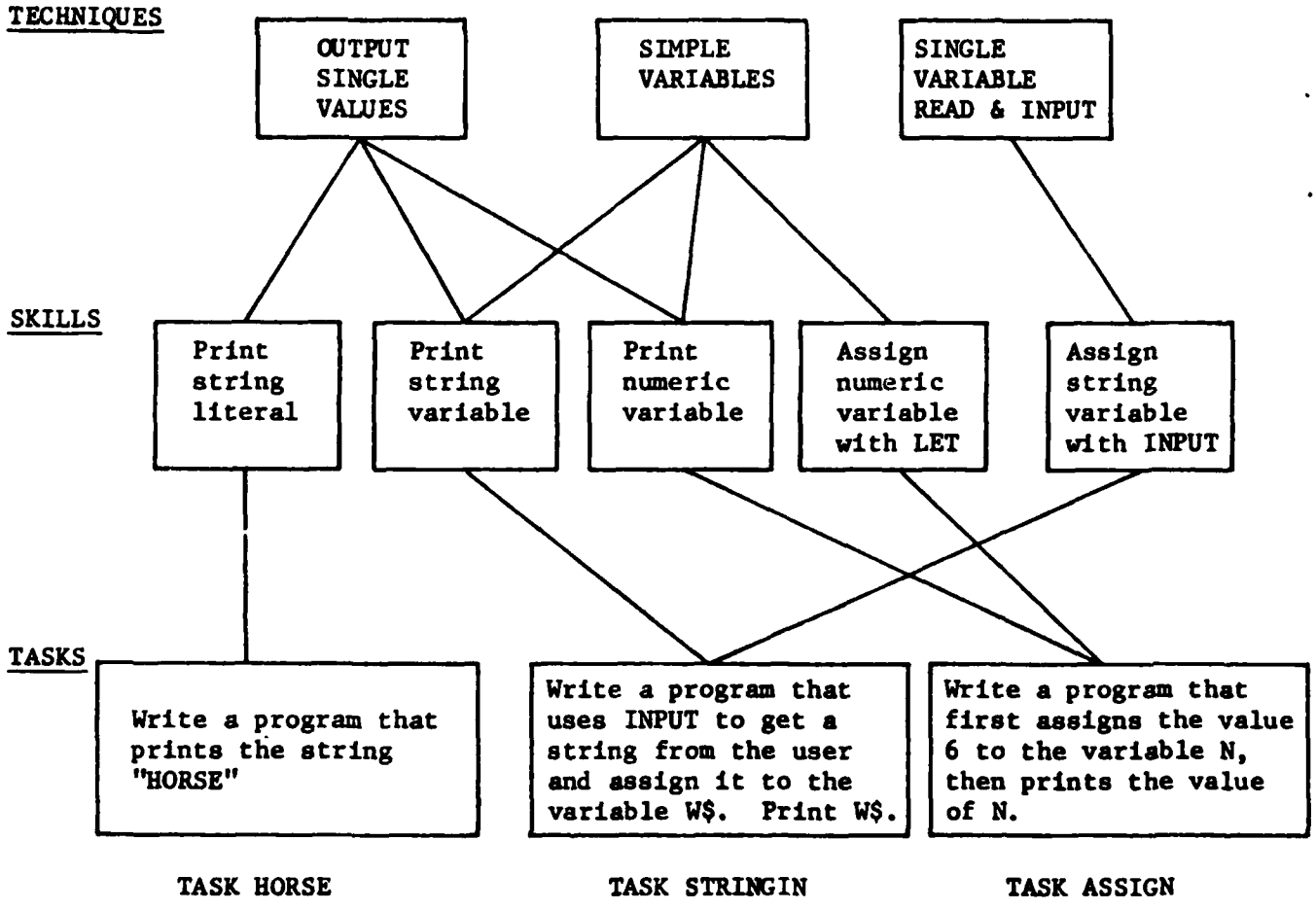


Figure 2. A simplified portion of the curriculum network.

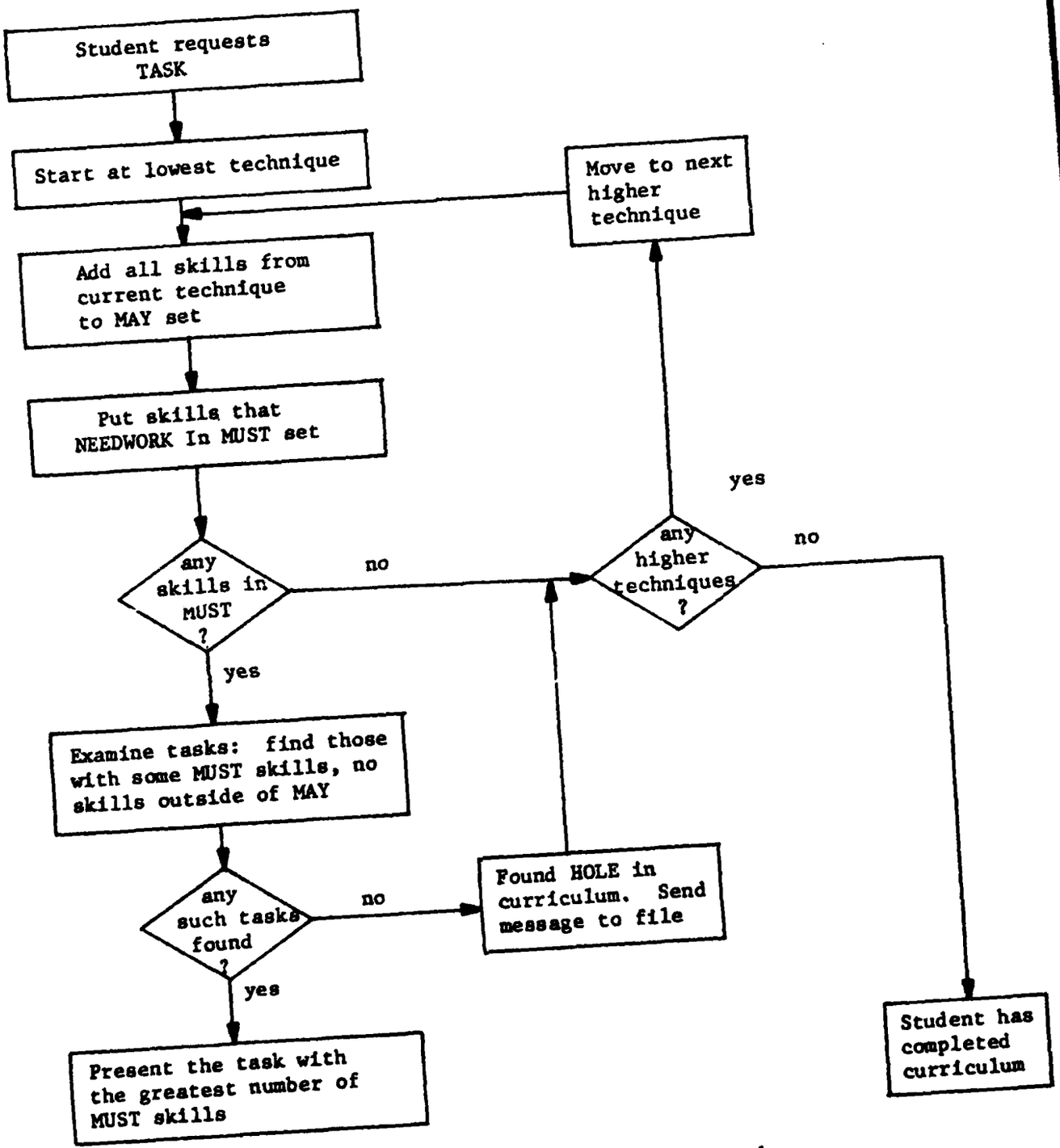


Figure 3. Selecting the next task.

If no MUST skills are found (indicating that the student has mastered all the skills at that technique level), the search process moves up by one technique, adding all its skills to the MAY set, then seeking MUST skills again. Once a MUST set is generated, the search terminates, and all of the tasks are examined by the procedure select. Those considered as a possible next task for the student must require (a) at least one of the MUST skills, and (b) no skills outside of the MAY set. Finally, the task in this group that requires the largest number of MUST skills is presented as the next task. Thus, in the simplified scheme shown in Figure 2, assuming that the student had not yet met the criterion on the skills shown, the first task to be presented would be HORSE, because its skill lies in the earliest technique, and would constitute the first MUST set. Task ASSIGN would be presented next, since its skills come from the next higher technique; STRINGIN would be presented last of these three.

An interesting curriculum development technique has evolved naturally in this scheme. If BIP has selected the MUST and MAY sets, but cannot find a task that meets the above requirements, then it has found a "hole" in the curriculum. After writing a message to the HOLES file (see Section 3.4 of Dageforde & Beard, Note 2) describing the nature of the missing task (e.g., the MUST and MAY skills), the procedure adjust examines the next higher technique. It generates new, expanded MUST and MAY sets, and then the procedure select again searches for an appropriate task. If none is found, a new search begins based on larger MUST and MAY sets. The only situation in which this process finally fails to select a task occurs when the student has covered all of the curriculum.

The first task a new student gets is not selected in this manner; it is automatically task GREENFLAG, which requires a two-line program solution. Because this is expected to be the student's first programming experience, and perhaps his first interaction of any kind with a computer, he is led through the solution to the task in very small steps. GREENFLAG is the only task in the curriculum that presents text (from the file GREENF), asks questions, and expects the student to type "answers," all of which alleviates the trauma of being told to write a program in the first session. However, since the student's responses are frequently commands that are passed to BIP's interpreter, he can see the effects of the input, and emerge from GREENFLAG having written and executed a genuine program.

When a student has finished GREENFLAG or any other task by successfully running his program, he proceeds by requesting "MORE." The procedure moreo first looks through the student's program for the BASIC operators and functions (if any) required in the solution to the task. If any required element is missing, he is asked to add it to the program and rerun it before again requesting "MORE." If the program contains all the required operators and functions, the procedure verify is called to evaluate it by comparing its output with that of the model solution run on the same test data (see Section 8), and the results are stored (in the student history) with each skill required by the task. Also, in the post-task interview, the student is asked to indicate whether or not he needs more work on the skills required by the task, which are listed separately. Thus, as mentioned above, BIP has two measures of the student's progress in each skill: its own comparison-test results, and the student's self-evaluation.

SECTION 8. VERFY MODULE

VERFY is the solution-checker module. Basically, it evaluates the student's program by comparing its output to that of the model solution. First, the procedure verify calls vinit to initialize variables, based on the coding line that precedes the model solution in the TASKS file. (See Section 5.2 of Dageforde & Beard, Note 2, for a detailed description of the coding line.) If that coding line does not start with a semicolon, then the student's program is not checked; it is assumed correct. Otherwise, the rest of the line is scanned for option code characters and value lists. If there is an "e," the variable wantExact is set to TRUE, since the "e" signifies that the student's program must produce the exact same number of output lines as that of the model. An "n" signifies that numeric expressions are to be stored for comparison (otherwise they are ignored), so the variable wantNumCon is set to TRUE. An "r" signifies that there are random numbers used in the programs, and the "random" numbers to be plugged in during the solution checker's invisible (to the student) execution of the model and of the student program are stored in the variable savrnd. An "s" signifies that string expressions are to be stored for comparison (otherwise they are ignored), so the variable wantStrCon is set to TRUE. A "v" signifies that all leading spaces should be discarded before output is saved for comparison, so the variable vacuum is set to TRUE.

Next, if there are any INPUT statements in the model (and thus presumably in the student program as well), haveInput is set to TRUE. The first few lines of the model solution are scanned, since for each INPUT statement, there must be a REM at the beginning of the model solution describing the use of the input variable. The format of each of those REM statements is:

<line #> REM <variable name> IS: <description> .

(For example: 10 REM X IS: THE USER'S FIRST ADDEND
20 REM Y IS: THE USER'S SECOND ADDEND)

A linked list of inputVars records is created, one for each INPUT variable expected. Each inputVars record has four fields: "name," for the variable name; "vals," for the value(s) to be assigned to the variable during execution; "description," for the description of the variable; and "link," for the pointer to the next record. The "name" and "description" fields are assigned according to the information in the REM statements, and the "vals" field is assigned by the appropriate value lists given at the end of the coding line.

After the initialization, the model solution is executed (invisibly), and every line of output is stored (in the array printl) for comparison, with the following exceptions. Any expression containing a quoted string or a numeric constant will not be stored, unless wantStrCon or wantNumCon, respectively, is TRUE. If vacuum is TRUE, all leading spaces are deleted before a line is stored. The integer variable outl tells the number of lines of model output stored.

If there is an execution error during the execution of the model, then the student is told so, and his program is assumed correct. Otherwise, the solution checker prepares to execute the student's program and compare its

output to that of the model. If there are any INPUT variables (if haveInput is TRUE), then the procedure kidputs is called to find out what variable names the student used. Kidputs goes through the linked list of inputVars records and asks the student "What variable do you use for ...?", where the "..." is replaced by the "description" field of the record. The variable name typed in by the student is checked to make sure that it is a valid variable name, that it is of the correct type (numeric or string), and that the student has not already said he used that variable name. If it is invalid for any reason, the student is told why and asked to retype the variable name. When a valid name is typed, it is compared to the "name" field of the record. If they are not the same, then the "name" field is replaced by the student's variable name. Once again, as before execution of the model solution, the "vals" field is set to the values to be assigned by INPUTs of this variable.

After finding out the student's input variable names, the elements of a Boolean array called matchup, which is parallel to the array printl, are initialized to FALSE. Then the student's program is executed (invisibly) and given the same "random" numbers and INPUT values (if any) as the model solution. Each line of its output (with the same vacuum, wantStrCon, and wantNumCon restrictions as for the model solution) is stored in the array printk and compared to the stored output from the model (printl). If the line matches the ith element of printl, then matchup[i] is set to TRUE. The variable outk tells the number of lines of student output stored. If, after the student's program has completed execution, any of the elements in the model-output array (printl) have not been matched (i.e., if matchup[i] = FALSE for any i, i = 1,2,...,outl), he is told that the program "does not seem to solve the problem," and the unmatched elements are listed. Or if wantExact is TRUE and outk is greater than outl, the student is told that the program produced too much output. In either case, the procedure verOption is called to determine whether or not the student wants to continue work on the task, and, if not, whether or not he disagrees with the solution checker. (If this is the case, this fact and the student's program are recorded in the file ARGUE). In addition to listing the output not found, verOption tells what values were assigned to INPUT variables, and which types of constants (numeric and/or string), if any, the solution checker ignored. If, on the other hand, all the model outputs have been matched (and outk = outl if wantExact is TRUE), the student is told that the program "looks ok," and the post-task interview (see Section 7) is presented.

SECTION 9. MSRECS MODULE

There are a number of linked lists of records built up and used throughout a BIP session. In particular, there is a linked list of records for every type of variable (numeric, string, one-dimensional numeric array, two-dimensional numeric array, one-dimensional string array) used in the student's current program. There is also a linked list of records for the user-defined functions, and the solution checker utilizes a list of the expected INPUT variables.

The first two fields of all these records are the same: "name" for the variable (or function) name, and "link" for a pointer to the next record in the linked list. In addition, the numeric and string variable records have a "val" field holding the current value (numeric or string) of the variable; the one-dimensional array variable records (numeric and string) have an "upperBound" field for the upper bound of the array (the lower bound of all BIP arrays is 1) and an "array a" field for the actual array. The two-dimensional numeric array records have the same fields as the one-dimensional ones plus a "secondUpperBnd" field for the upper bound of the second dimension of the array. INPUT variable records have a "vals" field for the values to be successively assigned to the INPUT variable by the solution checker. Finally, user-function records have an "index" field for the index into the argz array (see Section 4) of the line in the program containing the function definition.

Further examples of the extensive use of linked lists of records are:

1. The data structure for the main tasks (as described in the documentation for ONOFF) is a linked list of records, each with 13 fields.

2. The MUST and MAY "sets" of skills used in task-selection (see Section 7) are actually separate linked lists of records, each with two fields: "skIndex" for the skill number, and "link" for a pointer to the next record in the list.

3. The MAYBE "set" of possible next tasks to be presented by the task-selection algorithm is a linked list of records, each with three fields: "task" for a pointer to the task (in the linked list of main task records), "numMustSkills" for the number of skills in the MUST set that the task uses, and "link" for a pointer to the next record.

The MSRECS module contains procedures useful for handling records and linked lists of records. The "is" procedures (isnal, issa, etc.) check (at execution time) to see if a given variable is in the appropriate list (e.g., the one for one-dimensional numeric array variables, or the one for string variables). The "create" procedures (nvCreate, nalCreate, etc.) create a record of the desired class and initialize various fields according to the information passed to the procedures.

PutInMaybe adds a task to the MAYBE "set" described above. RemoveFromMaybe eliminates inappropriate tasks (those with skills outside the MAY "set") from that list.

ReqCreate creates a new record to be added to the linked list of operators or functions required in the solution to the current task. Each record has two fields, "name" for the name of the operator or function that is required, and "link" for a link to the next record in the list. ReqRemove removes a record from the list, isRequired checks to see if a particular operator is required (i.e., in the linked list) and listRequired simply steps through the list and types out the "name" field of each of the records (see Section 2).

SkCreate creates a new skill record and initializes its two fields: "skIndex" for the skill number, and "link" for a pointer to the next skill record in the list. AddtoList adds a new skill record to a linked list of skills (note that they are in numeric order, by skill number), and empty tells whether or not a particular skills list is empty. As mentioned above, the MUST and MAY "sets" used in task-selection are both linked lists of skills.

Insert inserts a record at the beginning of a linked list. Inlist checks to see whether or not a particular string is the same as the "name" field of any of the records in a specific linked list. All of the "is" procedures call inList with parameters telling which linked list (the one for numeric variables, or the one for string variables, etc.) to check and what "name" to search for.

REFERENCES

Barr, A., Beard, N., & Atkinson, R. C. The computer as a tutorial laboratory: The Stanford BIP project. International Journal of Man-Machine Studies, 1976, 8, 567-596.

Beard, M., Barr, A. V., Gould, L., & Wescourt, K. Curriculum information networks for computer-assisted instruction (NPRDC Tech. Rep. 78-18). San Diego: Navy Personnel Research and Development Center, April 1978.

Reiser, J. SAIL user's manual (Artificial Intelligence Memo 289). Stanford, CA: Stanford Artificial Intelligence Laboratory, Stanford University, 1976.

Wilcox, C. R. MAINSAIL language reference manual. SUMEX Computer Project, Stanford University Medical Center, 1977. (a)

Wilcox, C. R. The MAINSAIL project: Developing tools for software portability. Proceedings of the First Annual Symposium on Computer Application in Medical Care, IEEE Catalog No. 77CH1270-8C, pp. 76-83, Washington, D. C., 1977. (b)

REFERENCE NOTES

1. Dageforde, M. L. The BASIC Instructional Program: Conversion into MAINSAIL Language (NPRDC Tech. Note 78-11). San Diego: Navy Personnel Research and Development Center, April 1978.

2. Dageforde, M. L., & Beard, M. The BASIC Instructional Program: Supervisor's Manual (NPRDC Tech. Note 78-10). San Diego: Navy Personnel Research and Development Center, April 1978.

3. Beard, M. H., & Barr, A. V. The BASIC Instructional Program Student Manual (NPRDC Special Rep. 77-2). San Diego: Navy Personnel Research and Development Center, October 1976.

4. Dageforde, M. L., Beard, M. H., & Barr, A. V. The BASIC instructional program student manual: MAINSAIL conversion (NPRDC Special Rep. 78-9). San Diego: Navy Personnel Research and Development Center, April 1978

APPENDIX

LIST OF TECHNIQUES AND SKILLS IN TECHNIQUES

LIST OF TECHNIQUES AND SKILLS IN TECHNIQUES

Technique 1. Simple output—first programs.

1 Print numeric literal
2 Print string literal
5 Print numeric expression [operation on literals]
8 Print string expression [concatanation of literals]

Technique 2. Variables—assignment.

3 Print value of numeric variable
4 Print value of string variable
6 Print numeric expression [operation on variables]
7 Print numeric expression [operation on literals and variables]
9 Print string expression [concatanation of variables]
10 Print string expression [concatanation of variable and literal]
11 Assign value to a numeric variable [literal value]
12 Assign value to a string variable [literal value]

Technique 3. More complicated assignment.

34 Assign to a string variable [value of an expression]
35 Assign to a numeric variable [value of an expression]
69 Re-assignment of string variable (using its own value)
70 Re-assignment of numeric variable (using its own value)
82 Assign to numeric variable the value of another variable
83 Assign to string variable the value of another variable

Technique 4. More complicated output.

28 Multiple print [string literal, numeric variable]
29 Multiple print [string literal, numeric variable expression]
30 Multiple print [string literal, string variable]
74 Multiple print [string literal, string variable expression]

Technique 5. Interactive programs—INPUT from user—using DATA.

13 Assign numeric variable by -INPUT-
14 Assign string variable by -INPUT-
15 Assign numeric variable by -READ- and -DATA-
16 Assign string variable by -READ- and -DATA-
55 The REM statement

Technique 6. More complicated input.

17 Multiple values in -DATA- [all numeric]
18 Multiple values in -DATA- [all string]
19 Multiple values in -DATA- [mixed numeric and string]
22 Multiple assignment by -INPUT- [numeric variables]
23 Multiple assignment by -INPUT- [string variables]
24 Multiple assignment by -INPUT- [mixed numeric and string]
25 Multiple assignment by -READ- [numeric]
26 Multiple assignment by -READ- [string]
27 Multiple assignment by -READ- [mixed numeric and string]

Technique 7. Branching--program flow.

36 Unconditional branch (-GOTO-)
37 Interrupt execution

Technique 8. Boolean expressions.

38 Print Boolean expression [relation of string literals]
39 Print Boolean expression [relation of numeric literals]
40 Print Boolean expression [relation of numeric literal and variable]
41 Print Boolean expression [relation of string literal and variable]
75 Boolean operator -AND-
76 Boolean operator -OR-
77 Boolean operator -NOT-

Technique 9. IF statements--conditional standards.

42 Conditional branch [compare numeric variable with numeric literal]
43 Conditional branch [compare numeric variable with expression]
46 Conditional branch [compare two numeric variables]
47 Conditional branch [compare string variable with string literal]
48 Conditional branch [compare two string variables]
59 The -STOP- statement

Technique 10. Hand-made loops--iteration.

44 Conditional branch [compare counter with numeric literal]
45 Conditional branch [compare counter with numeric variable]
49 Initialize counter variable with a literal value
50 Initialize counter variable with the value of a variable
53 Increment the value of a counter variable
54 Decrement the value of a counter variable

Technique 11. Using loops to accumulate.

51 Accumulate successive values into numeric variable
52 Accumulate successive values into string variable
71 Calculating complex expressions [numeric literal and variable]
78 Initialize numeric variable (not counter) to literal value
79 Initialize numeric variable (not counter) to value of a variable
80 Initialize string variable to literal value
81 Initialize string variable to the value of another variable

Technique 12. Using "dummy" value to signify end of data.

20 Dummy value in -DATA- statement [numeric]
21 Dummy value in -DATA- statement [string]

Technique 13. BASIC functionals.

56 The -INT- function
57 The -RND- function
58 The -SQR- function

Technique 14. FOR...NEXT loops.

61 FOR . NEXT loops with literal as final value of index
62 FOR . NEXT loops with variable as final value of index
63 FOR . NEXT loops with positive step size other than 1
64 FOR . NEXT loops with negative step size

Technique 16. Arrays.

31 Assign element of string array variable by -INPUT-
32 Assign element of numeric array variable by -INPUT-
33 Assign element of numeric array variable [value is also a variable]
60 The -DIM- statement
65 String array using numeric variable as index
66 Print value of an element of a string array variable
67 Numeric array using numeric variable as index
68 Print value of an element of a numeric array variable

Technique 16. Nesting loops (one loop inside another).

72 Nesting loops
73 Subroutines (-GOSUB- and friends)

END

DTIC

4-86